Honest vs Cheating Bots in PATROL-based Real-Time Strategy MMOGs

Stefano Sebastio[1], Michele Amoretti[2], Jose Raul Murga[3],
Marco Picone[3], and Stefano Cagnoni[3]

1 : IMT Institute for Advanced Studies Lucca, Italy
{stefano.sebastio}@imtlucca.it
2 : Centro Interdipartimentale SITEIA.PARMA,
Università degli Studi di Parma, Italy
{michele.amoretti}@unipr.it
3 : Dip. di Ingegneria dell'Informazione,
Università degli Studi di Parma, Italy
{picone.m,josermurgav}@gmail.com
{stefano.cagnoni}@unipr.it

Abstract. The increasing success of massively multi-player online games (MMOGs) is due to the fact that they allow players to explore huge virtual worlds and to interact in many different ways, either cooperating or competing. To support the implementation of ultra-scalable real-time strategy MMOGs, we are developing a middleware, called PATROL, that is based on a structured peer-to-peer overlay scheme. Among other features, PATROL provides AI-based modules to detect cheating attempts, that the decentralized communication infrastructure may favor. In this paper we illustrate how we implemented honest and cheating autonomous players (bots). In particular, we show how honest bots can detect cheating bots in real-time, thanks to strategies based on neural networks.

1 Introduction

Most research on multi-player online games (MMOGs) focuses on scalability and high speed, but other issues such as the chance of cheating have an equally large practical impact on game success. There are significant technical barriers to achieving all these properties at the same time, and few existing games do so. Our open source middleware for the development of real-time strategy (RTS) MMOGs, called PATROL (http://code.google.com/p/patrol/), integrates several modules, each of which is highly specialized in one aspect of the game.

In our previous work [1] we focused on the module for peer-to-peer connectivity and communication, and on the module for detecting cheating behaviors. The former allows one to implement ultra-scalable RTS MMOGs, where each player's software installation is a node of a fully distributed structured overlay network scheme, which guarantees efficient data sharing, as well as fair and balanced workload distribution among participants. The latter, based on neural networks, allows each node to detect cheating behaviors of other players involved in the game.

In this paper we focus on the rule engine, a module that allows one to enforce both general and specific rules into the nodes. Game events can also be managed by the rule engine, by matching them with existing rules. Such a module allowed us to implement honest and cheating autonomous playing nodes. Such *bots* can play against each other, in a RTS game where participants place and move units and structures (generally speaking, *resources*) under their control to secure areas of the virtual world and/or destroy the assets of their opponents. We show how honest bots are able to detect misleading ones in real-time, thanks to the AI-based cheating detection module.

The paper is organized as follows. In section 2 we summarize some recent research work in the context of peer-to-peer RTS MMOGs. In section 3 we describe the PATROL architecture, with details on the rule engine and on the cheating detection strategy. In section 4 we illustrate the example of RTS MMOG we have implemented, based on the proposed architecture, where autonomous bots play against each other. We focus on the performance of the module for intelligent cheating detection, presenting and discussing many experimental results. Finally, in section 5, we conclude the paper by specifying plans for extending our work further.

2 Related Work

MMOG needs a messaging infrastructure for game actions and player communication. To this purpose, possible paradigms are Client-Server (CS), Peer-to-Peer (P2P), Client-Multi-Server (CMS), or Peer-to-Peer with Central Arbiter (PP-CA) [2]. Each solution has pros and cons, with respect to robustness, efficiency, scalability and security. In particular, when the architecture of the game is decentralized (*e.g.* P2P), facing malicious behaviors to support a large number of players is particularly challenging.

In [2], the authors propose a Mirrored-Arbiter (MA) architecture that combines the features of CMS and PP-CA. This architecture provides all the benefits of PP-CA, but also solves the main problems in PP-CA by using interest management techniques and multicast. Clients are divided into groups, each group being handled by an arbiter that maintains a global state of the game region and takes care of the consistency issue. When the arbiter receives an update from a client which conflicts with its game region state, it ignores the update and sends the correct region state to all clients in the group. The authors implemented a multiplayer game called ”TankWar” to validate the design of the proposed MA architecture. In our opinion, such a scheme is complex in the decision of the arbiters and their group assignments, and does not guarantee high scalability and security. Indeed, an arbiter may be a cheating node itself, which compromises the game for a large number of nodes.

In [3], the authors present a Peer-to-Peer (P2P) MMOG design framework, Mediator, based on a super-peer network with multiple super-peer (Mediator) roles. In this framework, the functionalities of a traditional game server are distributed, capitalizing on the potential of P2P networks, and enabling the MMOG to scale better with respect to both communication and computation. Mediator integrates four elements: a reward scheme, distributed resource discovery, load management, and super-peer selection. The reward scheme differentiates a peer's contribution from their reputation, and pursues symmetrical reciprocity as well as discouraging misdemeanors. The authors suggest to adopt the EigenTrust reputation management algorithm [4] and the DCRC anti-free-riding algorithm [5] as possible implementations for the reward scheme. Unfortunately, such schemes are complex and bandwidth-consuming.

The other aspect on which our paper focuses is the game engine. We believe that games are made of three elements: nouns (*i.e.* elements of the game, and variables related to them), verbs (the actions that players and player stand-ins can enact), and rules (limiting the nature of the existence of the nouns and creating relationships and interactions between them; limiting also which verbs can be enacted, when and in which context). Current programming paradigms do not provide an appropriate language for expressing these structures. Object-Oriented Programming (OOP) is very good at representing different types of objects ("nouns") and the relationships they may have between each other, with minimal duplicated code or wasted work. Unfortunately, OOP dictates that each class must encapsulate methods, *i.e.* actions that are strictly related to the class itself. Thus, OOP is not suitable for expressing verbs and rules as entities separated from nouns. In general, imperative languages (that are mostly used in game programming) are not good for clearly expressing verbs and rules. Instead, declarative languages based on first-order logic, like Prolog, are much more suitable.

Currently, one of the best known rule engines is Drools Expert [6], that uses the rule-based approach to implement an Expert System and is more correctly classified as a Production Rule System. A Production Rule System is Turing complete, with a focus on knowledge representation to express propositional and first order logic in a concise, non-ambiguous and declarative manner. The core of a Production Rules System is an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data against Production Rules — also called Productions or just Rules — to infer conclusions which result in actions. A Production Rule is a two-part structure that uses First Order Logic for reasoning over knowledge representation. There are a number of algorithms used for Pattern Matching by Inference Engines including Linear, Rete, Treat, Leaps. Drools implements and extends the Rete algorithm; Leaps used to be provided but was retired as it became unmaintained. While Drools Expert is a sound product, it is quite large and cannot be installed on portable devices. For this reason, we decided to adopt tuProlog [7], as discussed in next section.

3 PATROL middleware

In order to increase security, the game infrastructure should properly manage the interaction events among nodes. In RTS games, the most frequent events are those for: i) moving resources, ii) receiving updates about the virtual world, and iii) submitting the attacks. Our PATROL middleware manages these events through protocols that are appropriate for maintaining an adequate level of efficiency and security.

PATROL provides the following modules (illustrated in figure 1):

- Rule Engine
- Cheating Detector
- Overlay Manager
- GUI/GamePeer Connector

Since the Overlay Manager and the Cheating Detector have been already described in details in our previous work [1], here we just recall them shortly and we devote more space to the Rule Engine. The GUI/GamePeer Connector decouples the (game-specific) GUI from the GamePeer, which integrates the three previously listed general-purpose modules. For lack of space, we omit its description, but we emphasize that GUI decoupling also allows to implement games for mobile devices, where only the visualization may be running locally, while most computation processes may be executed remotely.

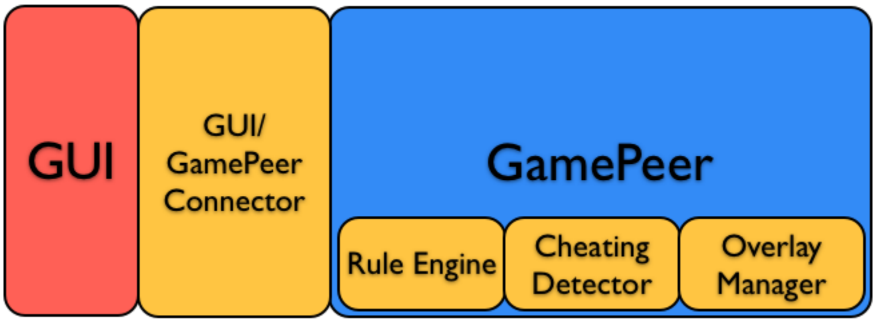

Fig. 1. A PATROL-based gaming node: PATROL modules are those in light color.

3.1 Overlay Manager

PATROL's Overlay Manager adopts the Chord P2P overlay scheme [8] to support fair and robust information sharing among available players. Chord is a highly structured P2P architecture where all peers are assigned the same role and amount of work. It is based on the DHT approach for an efficient allocation and recovery of resources. The overlay network in PATROL also supports a distributed algorithm for cheater detection, based on feedbacks among peers and AI tools such as neural networks. This approach allows one to dynamically recognize malicious behaviors, collectively performed by peers without the need of specific and centralized control components.

Chord [8] is probably the best known peer-to-peer protocol based on the Structured Model (SM), which uses DHTs as infrastructures for building large scale applications. Data are divided into *blocks*, each identified by a unique *key* (a hash of the block's name) and described by a *value* (typically a pointer to the block's owner). Each peer is assigned a random ID in same space of data block keys, and is responsible for storing key/value pairs for a limited subset of the entire key space.

According to Chord's lookup algorithm, each node n maintains a routing table with up to m entries, called the *finger table*. The i^{th} entry in the table at node n contains the identity of the first node s that follows n by at least 2^{i-1} on the identifier circle; i.e. $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$ and all the arithmetic is module 2^m. We call node s the i^{th} *finger* of node n, and denote it by $n.finger[i]$. A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node. Figure 2 illustrates the scalable lookup algorithm based on finger tables. In general, if node n searches for a key whose ID falls between n and the successor of n, node n finds the key in the successor of n; otherwise, n searches its finger table for the node n' whose ID most immediately precedes the one of the desired key, and then the basic algorithm is executed starting from n'. It is demonstrated that, with high probability, the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$ [8].

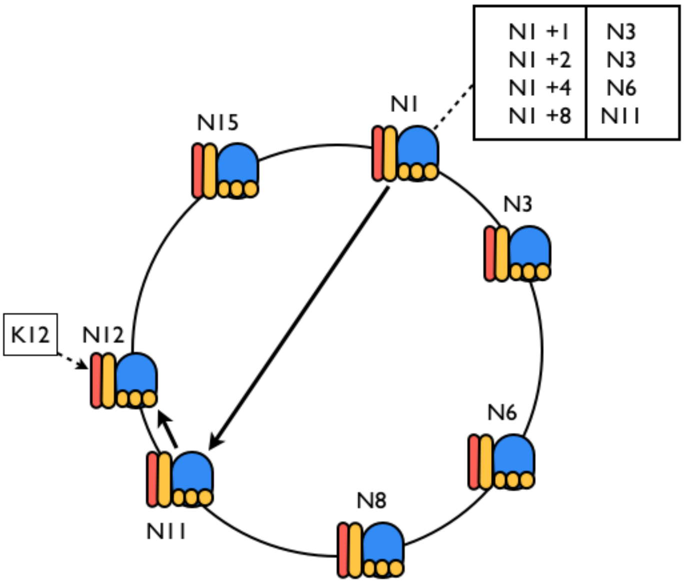

Fig. 2. The finger table entries for node $N1$ and the path taken by a query from $N1$, searching for key $K12$ using the scalable lookup algorithm.

PATROL distributes uniformly among the peers the responsibility to maintain knowledge about resources (*i.e.* items, war units, structures, etc.), using the DHT to share information about whom is responsible for what (each peer is responsible for a subset of the key space). In a game, each existing resource has a position in the virtual world. Such a position is hashed, and the resulting key is assigned to the peer whose key subset includes the resource key. It is very unlikely that two resources that are close in the virtual world have keys that are also close in the key space (and viceversa). Moreover, Chord foresees data replication, in order to improve robustness against unexpected node departures.

Importantly, no peer has full control over a region of the game space (like, for example, in [2]), so the damages that a hacked peer can do are very limited. Moreover, our approach is much more robust than existing decentralized solutions, because the departure of a node does not affect the games too much, thanks to the proactive data replication dictated by the Chord protocol.

3.2 Rule Engine

In general, a rule engine is a software system that, depending on the context, decides which rules to apply, and computes the result of their application, that may be a new knowledge item, or an action to perform. Usually, a rule engine includes the following components:

- a *rule base*, containing production rules whose structure is WHEN ⟨conditions⟩ THEN ⟨actions⟩;
- a *knowledge base*, also known as *work area*, that contains known facts;
- an *inference engine*, for processing rules.

Rules operate on facts of the knowledge base, that is dynamic as it can change over time, with new facts being added, and old facts being removed. Conditions of production rules are evaluated against facts. If a condition is true, the resulting action is the insertion of a new fact in the knowledge base, and possibly an action on the environment (*e.g.*, an action within the game).

PATROL's Rule Engine (from now on, RE) can be used for implementing several RTS MMOG: it is sufficient to set the appropriate rule base and knowledge base. Rules and facts must be written in Prolog, chosen because of its intuitiveness. RE's inference engine is based on tuProlog, a Java-based lightweight Prolog reasoner for Internet applications and infrastructures [7]. tuProlog provides a straightforward API to implement simple or more complex Prolog programs within Java code, or to read existing Prolog expressions from a file or from a database. Once one or more Prolog *theories* (*i.e.* ensembles of rule base and initial knowledge base) have been acquired, it is possible to use them to evaluate facts and derive new facts.

The RE can be used to decide which actions are allowed to the player, depending on his/her state and on the state of the game. Moreover, the RE can be used to implement bots, whose purpose is to allow real players to test their strategies before entering a game against other real players. A bot must be able to make decisions in all typical RTS situations, such as: resource accumulation, resource purchasing or building, resource improvement, displacement of mobile resources, attack against an enemy's resources, defense from an enemy's attack, goal checking.

The RE includes a PrologEngine class that provides methods for setting and managing a theory, and for solving queries. Such a class can be specialized (by means of inheritance) into different classes, each referring to an aspect of the game. Such specialized classes can be reused with different theories, and within different RTS MMOGs.

Game Events The system uses a bootstrap server to support peers in joining the network (which includes authentication, as well as Chord initialization) and configuring themselves for entering a game. In this way the bootstrap server has control over the accounts of the players and consequently provides a basic level of security.

Information about the virtual world may not be granted indiscriminately to any peer. Each peer has its own resources, which are placed in different positions of the virtual world, and has the right to receive information that refers to areas that are within the field of view of such resources, according to the rules of the game. Periodically, each peer needs to update its view on the virtual world. To do so, it sends specific requests to peers that are responsible for the positions that are visible. Before responding to such a request, peer j, that is responsible for position (x, y, z), checks his cache for updated information, and sends a request to verify the credentials for peer k. If everything is ok, it finally sends the response message.

Before performing any action that involves a change of game state, players must submit a request to the responsible of the resource that is affected by the action. For example, suppose player k can select a resource to be displaced in the virtual world; to perform the action "displacing resource to position (x, y, z)" the peer must submit the request to the node responsible for the key resulting from the hash of that position, *i.e.* $h(x, y, z)$. The peer that must become the new responsible for the displaced resource of peer k searches for the manager of the resource's current position (declared by peer k). Such peer is discovered by means of the hash of the current position. Thus the old manager checks in its cache whether it has the information on peer k and whether this information corresponds to what was declared to j. If the check is successful, peer j can decide, according to the game rules and considering the time elapsed between the changes of game state following the transition between the two positions, if it can accept the move and execute it, becoming the new responsible for the resource. If the position declared by k is not true, the request for resource displacement submitted by peer k is ignored and the state of the game remains the same.

While troops can be moved asynchronously by each player, attacks must be either asynchronous or synchronous (*i.e.* with a turn-based approach). In case of synchronous attacks, knowing the decisions of other players before submitting his own move may be a considerable advantage for a player. But, of course, this would be unfair.

To avoid cheating, PATROL uses request hashing, a mechanism that is widely adopted in other P2P architectures and derives from distributed security systems. Players who submit their decisions have to send a hash of the message describing the attack concatenated with a *nonce*. The nonce is used to prevent a cheater from storing in a table all matches between hash values and attack decisions, revealing the decisions of honest players. The nonce is a use-once random value, chosen by the first player that submits a decision. The last player that submits its decision may send it manifestly. At this point, all players that have previously sent the hash must send their nonce and attack decision manifestly. Thus, other players can re-calculate the hash and verify that it corresponds to what was previously declared. The properties of hash functions guarantee that it is almost impossible for two different attacks to have the same hash, and therefore for a player to submit a different attack, with respect to the encrypted one.

3.3 Intelligent Cheater Detection

PATROL provides a good level of security for the overall state of the game. However, the DHT does not prevent the game from offering cheaters (provided with hacked clients) the possibility to alter the information for which they are responsible. In a RTS game, a modified client that saves a history of recent attacks and their outcomes may estimate the current level of resources available to other players and take advantage over them.

Using artificial intelligence techniques, a PATROL-based peer can detect anomalies in the behavior of other peers, compared to typical behavioral profiles, by means of temporal analysis of interaction events. Moreover, using the power of direct communication typical of P2P approaches, a peer may ask other peers their "opinion" about a given peer in order to improve the evaluation process.

Peer x calculates the probability $P\{y|x\}$ that peer y is cheating. Then x sends a request to peers that have interacted with y, in order to match their probabilities and understand whether y is considered to be a cheater: $P\{y|i\}$ $\forall i \neq x, y$. If the global probability exceeds a certain threshold, there is the option to contact other peers and the bootstrap server to promote a collective motion against the cheating player in order to ban him from future games. If all peers agree with the "Ban Proposal", peer y is gracefully disconnected from the Chord ring.

The artificial intelligence module analyzes all action events coming from an opponent. The opinions of the other players are requested only at the end of the local evaluation process, if the peer estimates a high probability of cheating. Of course, the peer must be careful since other peers may provide false reports related to their interactions with a given peer.

There are different strategies for a peer to learn from a sequence of events: sequence recognition, sequence playback, and temporal association. Among others, we focused on the following tools:

- *Multi Layer Perceptrons (MLPs)*
- *Time Delay Neural Networks (TDNNs)*
- *Back-Propagation Through Time (BPTT) learning algorithms*

that fit very well our needs. We analyzed their features in details in our previous work [1].

4 Experimental evaluation

We have extended the PrologEngine class of the RE into specialized classes, to implement a spatial RTS MMOG for testing purposes. The goal of the RTS MMOG is to find and conquest all the planets that are in the game space.

Players are provided with a mine resource that allows them to make money for buying two types of resources: defense and attack. The resources for attack (starships) are used to explore the virtual world, to the purpose of finding the planets and to tackle the starships of other players. Every resource has an associated velocity and a field of view. The resources for defense are used to protect the owned planets from incoming attacks of other players' starships.

Thus, we have implemented ExtractionEngine for managing the extraction of mineral resources of a planet, BuyResourceEngine for purchasing resources, MovementEngine for moving mobile resources (like starships), VisibilityEngine for deciding if a resource (*e.g.* a planet, or an enemy's starship) is visible to the player, GameEvolutionEngine for deciding next operation depending on current state (own state, and game state), GameEngine for checking if intermediate or final goals have been met.

Based on such engines, each bot passes through three different phases:

- resource accumulation
- space exploration
- planet conquest

These phases are repeated in an infinite loop. The time periods each bots spends in such phases are random variables A,E and C.

We have defined two different types of bot profiles: honest and cheater. The latter reproduces the behavior of a hacked client. It owns a mine which is five times as powerful as the others and more initial money. Moreover, it has a halved cycle decision period (*i.e.* it can take more decisions in the same time).

In fig.3, we report the distribution of honest and cheating bots' actions during the recorded matches. On the horizontal axis we have the time at which the action is performed, considering 0 as the start time of the game, while on the vertical axis there is the value associated to the used resource. Here we can note that cheater bots, thanks to the speed hack, prefer to first explore the space and to perform its actions later than the honest bots. Moreover, the value associated to their resources is higher since the more money is available, the more they can spend for buying higher-valued resources.

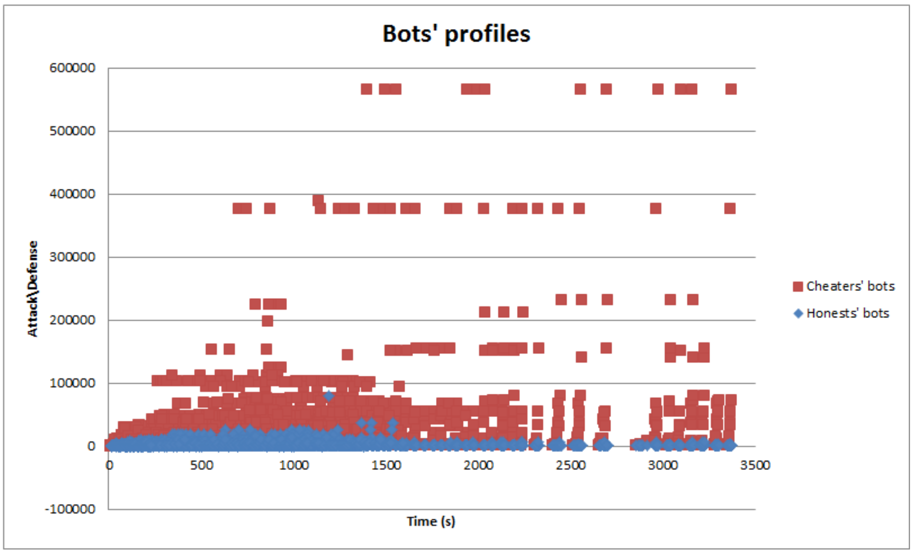

Fig. 3. Time distribution of bots' actions, considering all the games that have been played to collect a significant number of actions.

4.1 Performance of neural networks

We have collected 2800 player profiles. Each profile consists of a sequence of three actions performed by the opponent. The choice of the three moves for the profile duration appears as a good trade-off between quickness and accuracy of decisions about the opponent's behavior. The overall dataset has been split into two equally-sized sets, each including 700 honest and 700 cheater profiles. One set is used to train neural networks, and the other as a test set.

We have defined different configurations for the three types of neural networks that we took in consideration. Then we have compared these nets evaluating their *Root Mean Square Error (RMSE)*. After a series of preliminary tests, we observed that larger nets tend to overfit and are less able to generalize. This observation comes from a comparison between RMSE on the training and test sets that, in some case, shows that the performance on the test set is significantly worse than the peformance on the training set.

Finally, comparing the best performances achieved on the test set (in table 4.1) over a large number of attempts, we can see that the best performance is obtained by the TDNN, followed by BPTT and MLP. This result confirms that, not surprisingly, the neural networks that are specifically designed for analysis over time perform best.

Neural Networks	Free parameters	RMSE Test Set
MLP 6,3,1	21	0.4851
TDNN 6,6,15,1	117	0.4451
BPTT 6,7,15,1	133	0.4455

Table 1. RMSE values on the test set.

5 Conclusions

In this work we illustrated the most recent improvements of our PATROL framework for creating peer-to-peer online RTS games, characterized by a high degree of robustness, efficiency and effectiveness against cheating behaviors. In particular, we focused on the rule engine that allows to set manage the rules of a game, and also to develop autonomous virtual players (bots). We have shown how cheating bots can be detected by means of a PATROL module that uses neural networks.

Preliminary tests have been encouraging. In the future, we will perform a more accurate evaluation, considering different values of learning rate for the neural networks. Moreover, it is possible to envisage the use of other means of temporal analysis based on neural networks (such as *Real Time Recurrent Learning* and *Context Units* like Elman and Jordan nets) and on other techniques. It would also be possible to investigate the effects of adding a component capable of evaluating the *trust of peers* based on the past history of players.

References

1. M. Picone, S. Sebastio, S. Cagnoni, M. Amoretti, *Peer-to-Peer Architecture for Real-Time Strategy MMOGs with Intelligent Cheater Detection*, Proc. of DIstributed SImulation & Online gaming (DISIO), co-located with 3rd ICST/ACM International Conference on Simulation Tools and Techniques (SIMUTools 2010), Torremolinos, Spain, March 2010.
2. L. Yang, P. Sutinrerk, *Mirrored Arbiter Architecture - A Network Architecture for Large Scale Multiplayer Games*, Summer Computer Simulation Conference (SCSC 2007), pp.709-716, San Diego, California, USA, July 2007.
3. L. Fan, H. Taylor and P. Trinder, *Mediator: A Design Framework for P2P MMOGs*, Proc. of NetGames'07, pp.43-48, Melbourne, Australia, September 2007.
4. S. D. Kamvar, M. T. Schlosser and H. Garcia-Molina, *The EigenTrust Algorithm for Reputation Management in P2P Networks*, Proc. of the 12th Int'l Conf. World Wide Web, pp.640-651, Budapest, Hungary, May 2003.
5. M. Gupta, P. Judge, M. Ammar, *A reputation system for peer-to-peer networks.* Proc. of the 13th ACM NOSSDAV workshop, pp.144-152, Monterey, CA, USA, 2003.
6. JBoss Community, Drools home page, http://www.jboss.org/drools
7. E. Denti, A. Omicini, A. Ricci, *Multi-paradigm Java-Prolog integration in tuProlog*, Sci. Comput. Program., 57(2):217-250, 2005.
8. I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, Proc. of ACM SIGCOMM '01 Conference, pp.149-160, San Diego, USA, March 2001.